




More Recursive Backtracking

FRIDAY, APRIL 25

Further explorations in solving problems using recursive backtracking.

-  Readings: [Text](#) 9.1-9.3
-  [Lecture quiz on Canvas](#)
-  [Lecture video on Canvas](#)

Lecture Video

Click to sign in and play video

Contents

1. Problem Statement: 0-1 Knapsack (and Some Broken Solutions)
2. Similarity to Subset Generation
3. Preliminaries: Structs in C++
4. Test Cases
5. Overview: Why So Many Recursive Backtracking Solutions Today?
6. Recursive Backtracking Solution (1 of 3)
7. Recursive Backtracking Solution (2 of 3)
8. Recursive Backtracking Solution (3 of 3)
9. Best- and Worst-Case Runtimes for Recursive Knapsack Solutions
10. **Bonus Knowledge** Side Note: Inefficiency of the Solutions Above
11. Recursion and Backtracking Wrap
12. What's next?
13. Practice Problems

Problem Statement: 0-1 Knapsack (and Some Broken Solutions)

Suppose we have a sack with a weight capacity c , and we have items with weights (w_i) and values (v_i). The goal of

the 0-1 knapsack problem is to stuff items into our knapsack in such a way that maximizes the total value of the items we take (without exceeding the bag's maximum weight capacity). The problem is "0-1" because our decision about each item is binary: we either take it, or we don't. (There is a related problem called the ["continuous" or "fractional" knapsack problem](#) that allows us to take fractions of an item and is amenable to different solution from the one we're about to discuss.)

For example, suppose we have the following items, and a bag where $c = 10$:

Item 0: $w_0 = 4, v_0 = 6$

Item 1: $w_1 = 2, v_1 = 4$

Item 2: $w_2 = 3, v_2 = 5$

Item 3: $w_3 = 1, v_3 = 3$

Item 4: $w_4 = 6, v_4 = 9$

Item 5: $w_5 = 4, v_5 = 7$

Notice that if we start taking the most valuable items that we can hold, we won't maximize the value from our pillaging. We'll first take Item 4, then Item 5. Our total weight is then 10, and our total value is 16. However, there's a better solution. Do you see what it is?

Highlight for solution: Take items 1, 2, 3, and 5, for a total weight of $2 + 3 + 1 + 4 = 10$ and a total value of $4 + 5 + 3 + 7 = 19$.

You might object to the approach above and say that we should instead choose items based on (v_i / w_i) ratios (selecting the items with the highest ratios first). It turns out that will work for the example above, but here's a different example that shows where that approach *doesn't* work (again with max capacity $c = 10$):

Item 0: $w_0 = 8, v_0 = 160, v_0 / w_0 = 20$

Item 1: $w_1 = 3, v_1 = 58, v_1 / w_1 = 19.333$

Item 2: $w_2 = 3, v_2 = 58, v_2 / w_2 = 19.333$

Item 3: $w_3 = 3, v_3 = 58, v_3 / w_3 = 19.333$

Item 4: $w_4 = 1, v_4 = 2, v_4 / w_4 = 2$

Selecting by (v_i / w_i) ratios would cause us to take Item 0, leaving us only with room to take Item 4 after that. We would have a total value of 162 from those two items. The better solution here is to take the last four items, which give us a total weight of 10 and a total value of 176.

Similarity to Subset Generation

The recursive backtracking approach to the 0-1 knapsack problem is similar to the approach we have already seen for generating all subsets of a given set: for every element in a set, we made a binary choice about whether to include it in the subset we were generating or not, and we made two recursive calls accordingly -- one for each of those possibilities. Similarly, with 0-1 knapsack, we make binary choices about whether or not to place items from some input vector into our knapsack -- and make two recursive calls to explore those two distinct possibilities. A key distinction between the subset and knapsack algorithms is that in the knapsack algorithm, when we are faced with an item that is too heavy for our knapsack, we are forced to leave it behind -- and, accordingly, we make only one recursive call to reflect that. So, with the knapsack problem not *every* item in our input vector is subject to a binary choice.

Preliminaries: Structs in C++

I started building up to a solution by codifying the above test cases using two separate vectors, *weights* and *values*, like so:

```
PROVIDED_TEST("simple knapsack test")
{
    Vector<int> weights = {4, 2, 3, 1, 6, 4};
    Vector<int> values = {6, 4, 5, 3, 9, 7};

    int capacity = 10;

    EXPECT_EQUAL(knapsack(weights, values, capacity), 19);
}
```

The idea above is that our i^{th} item has its weight stored at `weights[i]` and its value stored at `values[i]`.

The idea of storing those values in two separate vectors should be somewhat concerning. What if we accidentally

perturb the elements in one vector, but not the other? In that event, the weights and values could become jumbled and mismatched. Our code would be far less error-prone if we could package the weight and value together in a single datatype.

In fact, C++ allows us to do that with something called a **struct**. A struct is effectively a new datatype that we articulate ourselves and which has the capability of holding multiple variables, which we refer to as the "members" or "fields" of the struct. The syntax for creating a struct is as follows:

```
struct TYPE_NAME
{
    DATA_TYPE MEMBER_NAME ;
    DATA_TYPE MEMBER_NAME ;
    ...
};
```

(*Common pitfalls!*) The semicolon after the closing curly brace above is required. Leaving off the semicolon is one of the most common errors with struct declarations. Note also that we declare our structs above and outside of the functions where we want to use the new datatype we're creating, or in a header (.h) file.

We created a `treasureT` struct capable of holding the weight and value of a single item:

```
struct treasureT
{
    int weight;
    int value;
};
```

(*Style note.*) People often place a `T` or `_t` at the end of a struct's type name so we can easily distinguish it from a variable name. (The "t" stands for "type," as in, "this isn't just the name of some variable -- it's a data type.")

`treasureT` is now a new datatype. It can be used to declare a new variable, like so:

```
treasureT myTreasure;
```

Above, `myTreasure` is now a variable of type `treasureT`. It has two "fields" or "members" that we can access using the dot operator like so:

```
myTreasure.weight = 4;
myTreasure.value = 6;
```

We could also initialize fields as follows, where values are assigned to fields in the order of those fields' declarations within our struct:

```
myTreasure = {4, 6};
```

In the example above, we get `myTreasure.weight = 4` and `myTreasure.value = 6`. (The 4 is assigned to the `weight` field because that field is declared first in the struct definition. The 6 is assigned to the `value` field because that one is declared second in the struct definition.)

Test Cases

With our newly created struct type, we threw down the following infrastructure before coding up our knapsack solution:

```

#include <iostream>
#include "console.h"
#include "vector.h"
#include "SimpleTest.h"
using namespace std;

struct treasureT
{
    int weight;
    int value;
};

// PLACE ONE OF THE KNAPSACK SOLUTION FUNCTIONS HERE

// Assumes weights and values vectors have equal size.
Vector<treasureT> createTreasureVector(Vector<int>& weights, Vector<int>& values)
{
    Vector<treasureT> treasures;

    for (int i = 0; i < weights.size(); i++)
    {
        treasureT myTreasure;
        myTreasure.weight = weights[i];
        myTreasure.value = values[i];
        treasures.add(myTreasure);
    }

    return treasures;
}

PROVIDED_TEST("simple knapsack test")
{
    Vector<int> weights = {4, 2, 3, 1, 6, 4};
    Vector<int> values = {6, 4, 5, 3, 9, 7};

    Vector<treasureT> treasures = createTreasureVector(weights, values);
    int capacity = 10;

    EXPECT_EQUAL(knapsack(treasures, capacity), 19);
}

PROVIDED_TEST("another simple knapsack test")
{
    Vector<int> weights = {8, 3, 3, 3, 1};
    Vector<int> values = {160, 58, 58, 58, 2};

    Vector<treasureT> treasures = createTreasureVector(weights, values);
    int capacity = 10;

    EXPECT_EQUAL(knapsack(treasures, capacity), 176);
}

int main()
{
    runSimpleTests(ALL_TESTS);
    return 0;
}

```

Overview: Why So Many Recursive Backtracking Solutions Today?

Having built out the test cases above, we began writing our backtracking solutions for the knapsack problem. In class, I presented three different solutions, each with their own advantages, disadvantages, and quirks. My goals in presenting multiple solutions to this problem today were:

- to reinforce the idea that there isn't just one "correct" recursive solution to each of the problems we've been exploring in this class
- to expose you to a variety of approaches in hopes of you finding one that really "clicks" and helps demystify recursion a bit
- to foster an exploration and discussion of the nuances of various solutions so that you can develop your own perspectives on what makes recursive functions more (or less) readable
- to help ensure that you won't be caught completely off guard when you see various types of recursive solutions outside of this class

(*Key take-away!*) The goal here is not for you to develop a masterful understanding of all three solutions below straight away. Rather, the goal is for you to develop a deep understanding of whichever one makes the most sense to you and to use that as a model for solving other problems recursively. There will be time after the quarter is over, or after you're more comfortable with recursion, to return to these examples and explore each of them more fully.

Recursive Backtracking Solution (1 of 3)

Below is the first backtracking solution we constructed for the knapsack problem. This one follows a common paradigm we've seen where we have a container (often called `rest`, but in this case called `treasures`) from which we remove elements as we descend deeper and deeper into our recursive calls. We hit our base case when the container we're choosing from is empty.

Note that this is a pass-by-reference solution, which means that we must add elements back to our vector after making our recursive calls for two reasons:

- Firstly, when we hit our first base case, our vector of treasures will be empty. If we never add anything back to that vector, then when we return to previous calls and start exploring other recursive branches of our problem space, there will never be any treasures to choose from. We need to add those treasures back so we can make our take-or-don't-take choices in other branches of the tree.
- Secondly, it's tremendously disruptive for us to destroy the contents of an input parameter over the course of solving some problem. The person who calls a function like this might still have use for the data they passed to us after the function is finished, so we need to be sure to restore values to the vector in question as we return our way out of this function.

With all that in mind, and with our binary choice approach in mind as well (take-or-don't-take), our first solution is as follows:

```
// Returns the maximum value we can derive by taking treasures from the given vector
// (without the total weight of the treasures we take exceeding the given capacity).
int knapsack(Vector<treasureT>& treasures, int capacity)
{
    // No more treasures to choose from means no more value to gain. Finito!
    if (treasures.size() == 0)
    {
        return 0;
    }

    // Here, we pull an item out of our vector in preparation for making a binary
    // choice about what to do with it: either take it or leave it behind.
    //
    // Removing from (and later adding to) the last position in our vector is more
    // efficient than operating on index 0 (zero). The latter requires us to scooch
    // over all the rest of the elements in the vector.
    treasureT thisOne = treasures[treasures.size() - 1];
    treasures.remove(treasures.size() - 1);

    int result;

    if (thisOne.weight <= capacity)
    {
        // Take it. Adding the current item to our knapsack adds to our running value
        // but also reduces the capacity of our knapsack.
        int with = thisOne.value + knapsack(treasures, capacity - thisOne.weight);

        // Don't take it. Leaving this item behind in hopes of finding other, better
        // items with which to fill our knapsack means we don't gain any value, but
        // we also don't reduce the capacity of our knapsack.
        int without = knapsack(treasures, capacity);

        result = max(with, without);
    }
    else
    {
        // Don't take it. This is the branch where thisOne is too heavy to take. So,
        // we don't have the luxury of making a binary choice about whether to take
        // the item with us. We must leave it behind.
        result = knapsack(treasures, capacity);
    }

    // Unchoose. See notes above this function about why we must add this back.
    treasures.add(thisOne);
    return result;
}
```

Recursive Backtracking Solution (2 of 3)

The second recursive solution we examined introduced a new variable, `k`, that started at 0 (zero) and worked its way up through our vector. I mentioned that with this sort of approach, there are often two ways to interpret the auxiliary integer variable, and that both interpretations effectively mean the same thing; they're just two sides of the same coin. In this particular example, `k` can be interpreted as follows:

- Within this call, we will consider what to do with the item at index `k` in the vector.

- We have already made decisions about what to do with the first k items in the vector.

If k starts at zero, then we will have made decisions about all items in the vector and hit a base case when $k == \text{treasures.size()} .$

A key advantage of this approach is that we don't need to do any of the bookkeeping associated with adding or removing items from our `treasures` vector. We still pass it by reference, which means we benefit from the time and space savings compared to passing it by value. The only downside here is that we need to develop a solid understanding of the purpose of that k variable.

The solution is as follows:

```
// Returns the maximum value we can derive by taking treasures from the given vector
// (without the total weight of the treasures we take exceeding the given capacity),
// focusing only on indices k and beyond.
int knapsack(Vector<treasureT>& treasures, int capacity, int k)
{
    // No more treasures to choose from means no more value to gain. Finito!
    if (k == treasures.size())
    {
        return 0;
    }

    // The structure below is very similar to that of today's first solution.
    // This time, however, we needn't remove items from the vector or add them
    // back in. The k variable helps us move through the vector without modifying
    // its contents.

    int thisWeight = treasures[k].weight;
    int thisValue = treasures[k].value;

    int result;

    if (thisWeight <= capacity)
    {
        // Take it. Adding the current item to our knapsack adds to our running value
        // but also reduces the capacity of our knapsack.
        int with = thisValue + knapsack(treasures, capacity - thisWeight, k + 1);

        // Don't take it. Leaving this item behind in hopes of finding other, better
        // items with which to fill our knapsack means we don't gain any value, but
        // we also don't reduce the capacity of our knapsack.
        int without = knapsack(treasures, capacity, k + 1);

        result = max(with, without);
    }
    else
    {
        // Don't take it. This is the branch where the current item is too heavy to take.
        // So, we don't have the luxury of making a binary choice about whether to take
        // the item with us. We must leave it behind.
        result = knapsack(treasures, capacity, k + 1);
    }

    return result;
}

// Wrapper function.
int knapsack(Vector<treasureT>& treasures, int capacity)
{
    // We kick off our journey by considering what to do with the element at index 0.
    // Another way to interpret this is that when we kick off our journey, we have so
    // far considered what to do with zero of the items in our vector.
    return knapsack(treasures, capacity, 0);
}
```

A more condensed version of this approach eliminated the `result` variable and simply returns from within the if-else blocks:

```
int knapsack(Vector<treasureT>& treasures, int capacity, int k)
{
    if (k == treasures.size())
    {
        return 0;
    }

    int thisWeight = treasures[k].weight;
    int thisValue = treasures[k].value;

    if (thisWeight <= capacity)
    {
        return max(thisValue + knapsack(treasures, capacity - thisWeight, k + 1),
                    knapsack(treasures, capacity, k + 1));
    }
    else
    {
        return knapsack(treasures, capacity, k + 1);
    }
}

int knapsack(Vector<treasureT>& treasures, int capacity)
{
    return knapsack(treasures, capacity, 0);
}
```

Recursive Backtracking Solution (3 of 3)

The third solution we examined was a twist on our first solution from today: it made both recursive calls ("with" and "without") rather than gating the "with" call behind a conditional statement that checked whether we could pick up the item in question. When making this change, I mentioned that Solution #1, with its conditional check on the weight of the item in question, was an example of "**arm's length recursion**," which is where we use a conditional statement to actively avoid making a recursive call in a certain situation rather than just making that call and allowing a base case to handle it for us. Accordingly, our modified solution needed to rely on the "with" call to detect -- in a base case -- whether we had picked up something that was too heavy for the knapsack.

To build toward a working solution with this modification, we need to check whether `capacity <= 0` as a base case for our recursive function. Assuming all of our items have positive weights (which seems reasonable), having `capacity == 0` would mean that we cannot take any additional items with us, and having `capacity < 0` would mean that we have picked up something too heavy for the knapsack and have to retreat from an infeasible path through our decision tree.

The big problem here is that if we have a line like this:

```
int with = thisOne.value + knapsack(treasures, capacity - thisOne.weight);
```

... then if our item was too heavy to pick up, our recursive call is effectively powerless to undo our addition of `thisOne.value` to the value of the recursive call we're making. (Indeed, if the item was too heavy to pick up, we do **not** want to add its value to any running total we're keeping track of.)

One way around that issue is to pass the total value of all the items we have picked up so far to the function as a third parameter, like so:


```

// Returns the maximum value we can derive by taking treasures from the given vector
// (without the total weight of the treasures we take exceeding the given capacity).
// This version assumes all item weights are positive.
int knapsack(Vector<treasureT>& treasures, int capacity, int valueSoFar)
{
    // If capacity is less than zero, we've tried to pick up something that is too
    // heavy to go into our knapsack, and we have therefore entered into an invalid
    // state. Thus, we derive no value from this state -- not even the valueSoFar.
    if (capacity < 0)
    {
        return 0;
    }

    // No more treasures to choose from means no more value to gain. Similarly,
    // no knapsack capacity means nothing else to gain, assuming all item weights
    // are positive. We return the valueSoFar, since there's nothing else to add.
    // Finito!
    if (treasures.size() == 0 || capacity == 0)
    {
        return valueSoFar;
    }

    treasureT thisOne = treasures[treasures.size() - 1];
    treasures.remove(treasures.size() - 1);

    // Take it. Adding the current item to our knapsack adds to our running value
    // but also reduces the capacity of our knapsack.
    int with = knapsack(treasures, capacity - thisOne.weight, valueSoFar +
thisOne.value);

    // Don't take it. Leaving this item behind in hopes of finding other, better
    // items with which to fill our knapsack means we don't gain any value, but
    // we also don't reduce the capacity of our knapsack.
    int without = knapsack(treasures, capacity, valueSoFar);

    // Unchoose. See notes above this function about why we must add this back.
    treasures.add(thisOne);
    return max(with, without);
}

// Wrapper function.
int knapsack(Vector<treasureT>& treasures, int capacity)
{
    // At the start of our journey, we haven't picked up any treasures yet, so our
    // valueSoFar (the third parameter below) is 0 (zero).
    return knapsack(treasures, capacity, 0);
}

```

Best- and Worst-Case Runtimes for Recursive Knapsack Solutions

In the worst case, every item is light enough to add to our knapsack in every recursive call we make, and so each call (aside from our base case) spawns two more recursive calls, leading to an exponential runtime, **$O(2^n)$** (where n is the size of our original vector of treasures). In the best case, every item is too heavy to add to our knapsack, and so each call we make (aside from our base case) spawns only one recursive call, leading to a linear runtime, **$O(n)$** .

Bonus Knowledge Side Note: Inefficiency of the Solutions Above

I feel I would be remiss if I did not point out that the recursive approaches above are **not** the most efficient solutions to 0-1 knapsack. In later courses, you will learn about two techniques -- memoization and iterative dynamic programming -- that can be applied to the recursive algorithms above to eliminate redundant recursive calls and drastically improve the worst-case runtime.

Recursion and Backtracking Wrap

This wraps up our intense, five-day introduction to recursion and recursive backtracking. While these topics are tricky when you first encounter them, I hope they're starting to click into place, and I hope you're starting to see how incredible it is that we can now solve problems in just a few lines of code that would be significantly more complex to solve without recursion.

What's next?

Monday is our first Celebration of Knowledge! Be sure to thoroughly review all the information on our [midterm exam page](#), as I don't want you to encounter any surprises on Monday. To give you additional time to prepare, there will be no lecture on Monday.

When we return to class next Wednesday, we shift gears to new topics: object-oriented programming and dynamic memory management. Those will serve as building blocks for our exploration of how ADTs are implemented behind the scenes and our examination of even more sophisticated data structures than the ones we have seen so far.

Practice Problems

1. Strengthen your understanding of recursion by tracing through the behavior of one of the knapsack solutions above on the first provided test case. Either construct a recursive tree diagram manually, or use the debugger to examine the behavior of the function from start to finish.
2. Take at least a 30-minute break from studying, then come back and try to implement a recursive knapsack solution from scratch. Focus on recreating the code from your understanding of how it works, not from memorization of the code itself.
3. An excellent question posed by someone after class this quarter (Spring '25): Instead of creating a struct, why can't we just use a map to track our weight-value pairs for knapsack inputs? The *weights* could serve as *keys* in such a map, and the *values* they would map to would be... well, the *values* of the items they represented. For example, suppose we had the following three items:

Item 1: $w_1 = 2$, $v_1 = 4$

Item 2: $w_2 = 3$, $v_2 = 5$

Item 3: $w_3 = 1$, $v_3 = 3$

We could put these in a map like so:

```
Map<int, int> treasures;  
map[2] = 4;  
map[3] = 5;  
map[1] = 3;
```

Where does this representation potentially break down?

4. Modify the first backtracking solution in today's notes to pass parameters by value rather than by reference. What operation becomes unnecessary when passing those parameters by value, and why?

Highlight for solution to Problem #4: When passing by value, we needn't add treasures back to the vector before returning from a recursive call. That's because the call we're about to return to has its own copy of the vector from before the treasure in question was removed.

5. As always, the textbook and this week's section handout are chock full of great exercises to reinforce this material, as well.

All course materials © Stanford University 2024. This content is protected and may not be shared, uploaded, or distributed.

Website programming by Julie Zelenski with modifications by Sean Szumlanski • Styles adapted from Chris Piech • This page last updated 2025-Apr-26